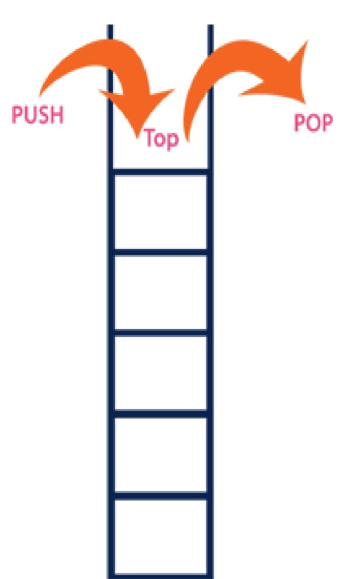
Stack

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at single position which is known as "top". That means, new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on LIFO (Last In First Out) principle.



In a stack, the insertion operation is performed using a function called "push" and deletion operation is performed using a function called"pop".In the figure, PUSH and POP operations are performed at top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)

A stack data structure can be defined as follows...

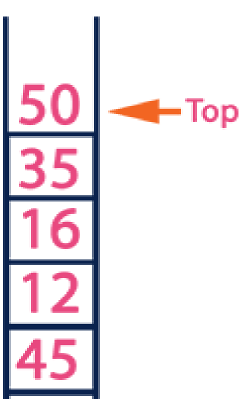
Stack is a linear data structure in which the operations are performed based on LIFO principle.

Stack can also be defined as

"A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".

Example

If we want to create a stack by inserting 10,45,12,16,35 and 50. Then 10 becomes the bottom most element and 50 is the top most element. Top is at 50 as shown in the image...



Operations on a Stack

The following operations are performed on the stack...

- **Push (To insert an element on to the stack)**
- **Pop (To delete an element from the stack)**
- **Display (To display elements of the stack)**

Stack data structure can be implement in two ways. They are as follows...

- Using Array
- Using Linked List

- When stack is implemented **using array**, that stack can organize only limited number of elements.
- When stack is implemented **using linked list**, that stack can organize unlimited number of elements.

Stack Using Array

A stack data structure can be implemented using one dimensional array. But stack implemented using array, can store only fixed number of data values. This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using LIFO principle with the help of a variable 'top'. Initially top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

A stack can be implemented using array as follows...

steps to create an empty stack.

Step 1: Include all the header files which are used in the program and define a constant **'SIZE'** with specific value.

Step 2: Declare all the functions used in stack implementation.
Step 3: Create a one dimensional array with fixed size **(int stack[SIZE])**
Step 4: Define a integer variable 'top' and initialize with '-1'. **(int top = -1)**
Step 5: In main method display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

Step 1: Check whether stack is FULL. **(top == SIZE-1)**
Step 2: If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!!" and terminate the function.
Step 3: If it is NOT FULL, then increment top value by one **(top++)** and set **stack[top]** to value **(stack[top] = value).**

pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from top position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

Step 1: Check whether stack is EMPTY. **(top == -1)**
Step 2: If it is EMPTY, then display "Stack is EMPTY!!! Deletion is not possible!!!" and terminate the function.
Step 3: If it is NOT EMPTY, then delete **stack[top]** and decrement top value by one **(top--).**

display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...
Step 1: Check whether stack is EMPTY. **(top == -1)**
Step 2: If it is EMPTY, then display "Stack is EMPTY!!!" and terminate the function.
Step 3: If it is NOT EMPTY, then define a variable 'i' and initialize with top.
Display **stack[i]** value and decrement i value by one **(i--).**
Step 4: Repeat above step until i value becomes '0'.

C++ Code for Array based Stack

```
#include<iostream.h>
#include<conio.h>

#define SIZE 10

void push(int;
void pop(;
void display(;

int stack[SIZE], top = -1;

void main()
{
   int value, choice;
   clrscr(;
   while(1){
("\n\n***** MENU *****\n”;
      cout<<"1. Push\n2. Pop\n3. Display\n4. Exit";
      cout<<"\nEnter your choice: “;
      cin>>choice;
      switch(choice){
     case 1: cout<<"Enter the value to be insert: ";
         cin>>value;
         push(value);
         break;
     case 2: pop();
         break;
     case 3: display();
         break;
     case 4: exit(0);
     default: cout<<"\nWrong selection!!! Try again!!!”;
      }
   }
}
void push(int value){
   if(top == SIZE-1)
      cout<<"\nStack is Full!!! Insertion is not possible!!!”;
   else{
      top++;
      stack[top] = value;
      cout<<"\nInsertion success!!!”;   }  }
```

```
void pop(){
   if(top == -1)
      cout<<"\nStack is Empty!!! Deletion is not possible!!!”;
   else{
      cout<<"\nDeleted : Data”<< stack[top];
      top--;    }
}
void display(){
   if(top == -1)
      cout<<"\nStack is Empty!!!”;
   else{
      int i;
      cout<<"\nStack elements are:\n”;
      for(i=top; i>=0; i--)
     cout<<"stack[i]<<”\t”;    }
}


//Object Oriented Code
#include <iostream.h>
#include <stdio.h>
struct student
{
  int roll;
  char name[20];
};
class stack
{
  int top, smax;
  student* array;
  public:
    stack(int n=5)
    {  array=new student[smax=n]; top=-1; }
    void push();
    void pop();
    void display();
    ~stack() { delete []array; }
};

void stack::push()
{
  if (top==smax-1)
    cout<<"Stack Overflow\n";
```

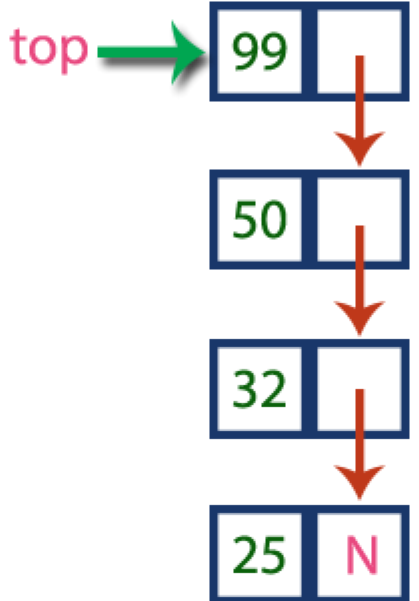
```
  else {
         student t;
         cout<<"Roll? "; cin>>t.roll;
         cout<<"Name? "; gets(t.name);
         array[++top]=t;
         cout<<t.roll<<' '<<t.name<<' '<<" pushed\n"; }
}
void stack::pop()
{
  if (top==-1)
    cout<<"Stack Underflow\n";
  else {
         student t=array[top--];
         cout<<t.roll<<' '<<t.name<<' '<<" popped\n";   }
}
void stack::display()
{
  if (top==-1)
    cout<<"Stack Empty\n";
  else{
         cout<<"Displaying Stack\n";
         for (int k=top; k>=0; k--){
              student t=array[k];
              cout<<t.roll<<' '<<t.name<<' '<<endl;}     }
}
void main()
{
  stack obj(10);
  int ch;
  do
  {
    cout<<"1. Push into Stack\n";
    cout<<"2. Pop from Stack\n";
    cout<<"3. Display Stack\n";
    cout<<"0. Exit\n";
    cout<<"Choice[0-3]? "; cin>>ch;
    switch (ch)
    {
      case 1: obj.push(); break;
      case 2: obj.pop(); break;
      case 3: obj.display(); break;
    }
  }while (ch!=0);   }
```

Stack using Linked List

The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as 'top' element. That means every newly inserted element is pointed by 'top'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by 'top' by moving 'top' to its next node in the list. The next field of the first element must be always NULL.

Example



In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

Operations

To implement stack using linked list, we need to set the following things before implementing actual operations.

Step 1: Include all the header files which are used in the program. And declare all the user defined functions.
Step 2: Define a 'Node' structure with two members data and next.
Step 3: Define a Node pointer 'top' and set it to NULL.
Step 4: Implement the main method by displaying Menu with list of operations and make suitable function calls in the main method.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...
Step 1: Create a newNode with given value.
Step 2: Check whether stack is Empty (top == NULL)
Step 3: If it is Empty, then set newNode → next = NULL.
Step 4: If it is Not Empty, then set newNode → next = top.
Step 5: Finally, set top = newNode.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...
Step 1: Check whether stack is Empty (top == NULL).
Step 2: If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!" and terminate the function
Step 3: If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.
Step 4: Then set 'top = top → next'.
Step 7: Finally, delete 'temp' (free(temp)).

display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...
Step 1: Check whether stack is Empty (top == NULL).
Step 2: If it is Empty, then display 'Stack is Empty!!!' and terminate the function.
Step 3: If it is Not Empty, then define a Node pointer 'temp' and initialize with top.
Step 4: Display 'temp → data --->' and move it to the next node. Repeat the same until temp reaches to the first node in the stack (temp → next != NULL).
Step 5: Finally! Display 'temp → data ---> NULL'.

```
//Procedural Oriented Code
#include <iostream.h>
#include <stdio.h>
struct node {
  int roll;
  char name[20];
  node* next; };
void main()
{
  node* top=NULL;
  node* p;
  int ch;
  do
  {
    cout<<"1. Push into Stack\n";
    cout<<"2. Pop from Stack\n";
    cout<<"3. Display Stack\n";
    cout<<"0. Exit\n";
    cout<<"Choice[0-3]? "; cin>>ch;
    switch (ch)
    {   case 1:
            p=new node;
        if (p==NULL)
            cout<<"Stack Overflow\n";
        else {
            cout<<"Roll? "; cin>>p->roll;
            cout<<"Name? "; gets(p->name);
            p->next=top;
            top=p;
            cout<<p->roll<<','<<p->name<<','<<" Pushed\n";  }
            break;
        case 2:
        if (top==NULL)
            cout<<"Stack Underflow\n";
        else  {
                p=top;
                top=top->next;
                cout<<p->roll<<','<<p->name<<','<<"
                popped\n";
                delete p;    }
                break;
      case 3:
        if (top==NULL)
```

```
            cout<<"Stack Empty\n";
        else {
                p=top;
                cout<<"Displaying Stack\n";
                while (p!=NULL) {
                  cout<<p->roll<<','<<p->name<<','<<endl;
                  p=p->next;    }       }
        }
    }while (ch!=0);
}
```

//Object Oriented Code

```
#include <iostream.h>
#include <stdio.h>
struct node {
    int roll;
    char name[20];
    node* next; };
class stack
{
  node* top;
  public:
    stack() { top=NULL; }
    void push();
    void pop();
    void display();
    ~stack();
};
void stack::push()
{
  node* p=new node;
  if (p==NULL)
    cout<<"Stack Overflow\n";
  else {
        cout<<"Roll? "; cin>>p->roll;
        cout<<"Name? "; gets(p->name);
        p->next=top;
        top=p;
        cout<<p->roll<<','<<p->name<<','<<" Pushed\n"; }
}
void stack::pop()
{
```

```
  if (top==NULL)
    cout<<"Stack Underflow\n";
  else  {
        node* p=top;
        top=top->next;
        cout<<p->roll<<','<<p->name<<','<<" popped\n";
        delete p; }
}
void stack::display()
{
  if (top==NULL)
    cout<<"Stack Empty\n";
  else {
        node* p=top;
        cout<<"Displaying Stack\n";
        while (p!=NULL) {
            cout<<p->roll<<','<<p->name<<','<<p->fees<<endl;
            p=p->next;  }     }
}
stack::~stack()
{
  while (top!=NULL){
        node* p=top;
        top=top->next;
        delete p;  }
}
void main()
{
  stack obj; int ch;
  do
  {
    cout<<"1. Push into Stack\n ";
    cout<<"2. Pop from Stack\n";
    cout<<"3. Display Stack\n";
    cout<<"0. Exit\n";
    cout<<"Choice[0-3]? "; cin>>ch;
    switch (ch)
    {
      case 1: obj.push(); break;
      case 2: obj.pop(); break;
      case 3: obj.display(); break;
    }
  }while (ch!=0); }
```

APPLICATION OF STACK

Infix notation.	*Prefix* notation.	Postfix notation
• To add A, B, we write **A+B** • To multiply A, B, we write **A*B** • The operators ('+' and '*') go in between the operands ('A' and 'B') • This is ***Infix*** notation.	• Instead of saying "A plus B", we could say "add A,B " and write **+ A B** • "Multiply A,B" would be written *** A B** • The operators ('+' and '*') go in before the operands ('A' and 'B') This is ***Prefix*** notation.	• Another alternative is to put the operators after the operands as in **A B +** and **A B *** • The operators ('+' and '*') go in after the operands ('A' and 'B') This is ***Postfix*** notation
Evaluate 2+3*5. + First: **(2+3)*5** **= 5*5 = 25** * First: **2+(3*5)** **= 2+15 = 17** • Infix notation requires Parentheses.	**+ 2 * 3 5** **= + 2 * 3 5** **= + 2 15** = 17 *** + 2 3 5 =** **= * + 2 3 5** **= * 5 5 = 25** • No parentheses needed!	**2 3 5 * +** = 2 **3 5 *** + **= 2 15 + = 17** **2 3 + 5 *** = **2 3 + 5 *** = **5 5 * = 25** • No parentheses needed here either
((A + B)*(C+D))	Move each operator to the LEFT of its operands & remove the parentheses: **((A + B)*(C+D))** **(+AB) *(+CD)** ***+AB+CD**	Move each operator to the RIGHT of its operands & remove the parentheses: **((A + B)*(C+D))** **(AB+) *(+CD)** **AB+CD+***

INFIX TO POSTFIX

RULES TO FOLLOW

- Print operands as they arrive.
- If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
- If the incoming symbol is a left parenthesis, push it on the stack.
- If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses
- If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
- If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
- At the end of the expression, pop and print all operators on the stack.

Convert following infix notation => (A + B) * C + D / (E * G) – H to Postfix

INPUT	STACK	POSTFIX	EXPLANATION
(	(		
(	((		**(** Left Parenthesis will go in stack
A	((	A	**A** (operand) will go to postfix
+	((+	A	**+** (operator) will be pushed to stack at top
B	((+	AB	**B** operand) will go to postfix after A
)	(	AB+	**)** Right Parenthesis will make all operator(s) till previous left parentheses to pop from top one by one and added to end of Postfix
*****	(*	AB+	***(operator) will be pushed to stack to top
C	(*	AB+C	**C** (operand) will go to postfix
+	(+	AB+C*	**+** (operator) will be pushed to stack topv after pop out * Operator from stack as it has higher precedence than +
D	(+	AB+C*D	**D** (operand) will go to postfix
/	(+/	AB+C*D	**/** (operator) will be pushed to stack at top
(	(+/(	AB+C*D	**(** Left Parenthesis will go in stack at top
E	(+/(	AB+C*DE	**E** (operand) will go to postfix
*****	(+/(*	AB+C*DE	***(operator) will be pushed to stack to top
G	(+/(*	AB+C*DEG	**G** (operand) will go to postfix
)	(+/	AB+C*DEG*	**)** Right Parenthesis will make all operator(s) till previous left parentheses will pop from top one by one and added to end of Postfix
-	(-	AB+C*DEG*/+	**-**(operator) will be pushed to stack to top
H	(-	AB+C*DEG*/+H	**H** (operand) will go to postfix
)	EMPTY	**AB+C*DEG*/+H-**	**)** Right Parenthesis will make all operator(s) till previous left parentheses will pop from top one by one and added to end of Postfix /As this make **Stack empty**, the current Postfix String is the resultant **POSTFIX NOTATION**

Evaluation of Postfix notation

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix.

Keep the following points for evaluation postfix expressions.
1) Create a stack to store operands (or values).
2) Scan the given expression and do following for every scanned element.
   - **If the element is a number, push it into the stack**
   - **f the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack**
3) When the expression is ended, the number in the stack is the final answer

Example

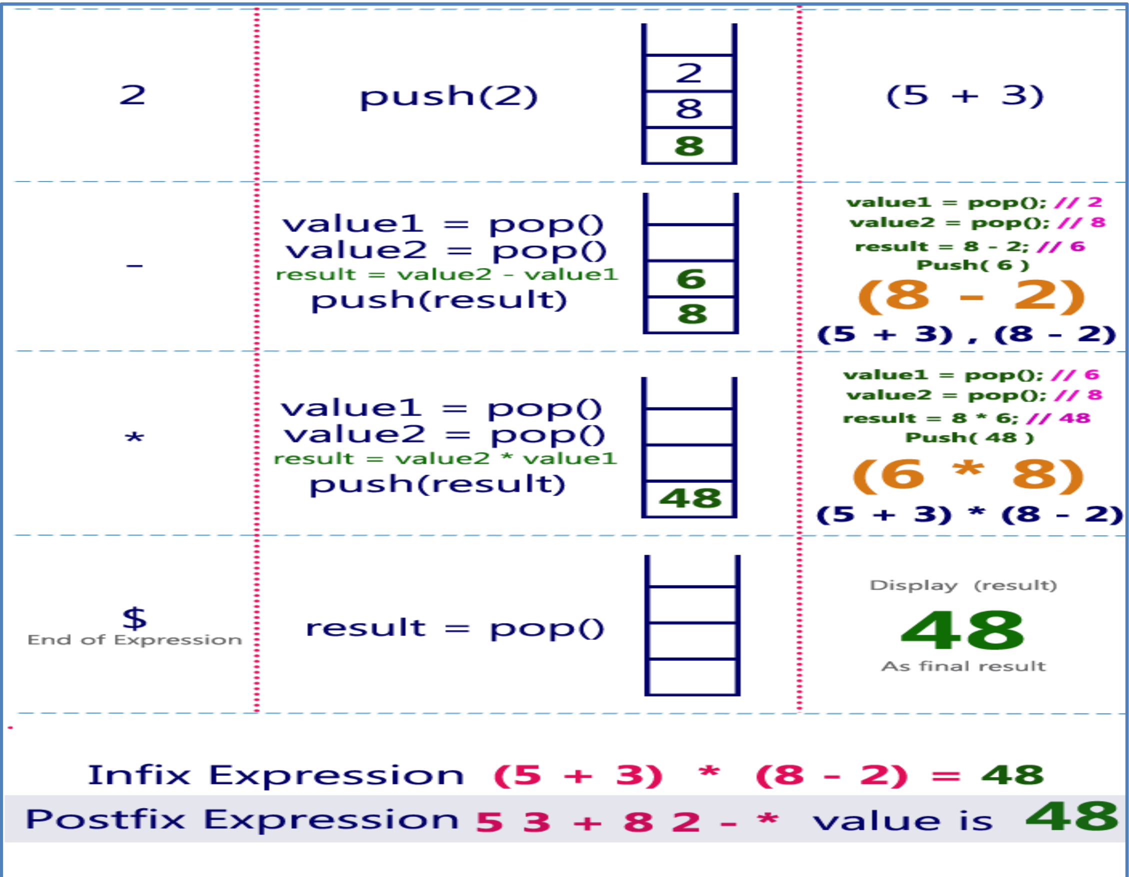
Infix Expression (5 + 3) * (8 - 2)
Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Stack	Evaluated Part of Expression
Initially	Stack is Empty		Nothing
5	push(5)	5	Nothing
3	push(3)	3, 5	Nothing

+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result)
		8	value1 = pop(); // 3
			value2 = pop(); // 5
			result = 5 + 3; // 8
			Push(8)
			(5 + 3)

8	push(8)	8, 8	(5 + 3)



Example 2

4 5 + 9 * 3 + 3 /

INPUT	ACTION	STACK
4	PUSH 4 TO STACK (operand are pushed to stack)	4
5	PUSH 5 TO STACK(operand are pushed to stack)	5 4
+	POP 5 AND POP 4 AND CALCULATE VALUE WITH OPERATOR (4+5) AND PUSH THE RESULT TO STACK	9
9	PUSH 9 TO STACK (operand are pushed to stack)	9 9
*	POP 9 AND POP 9 AND CALCULATE VALUE WITH OPERATOR (9* 9) AND PUSH THE RESULT TO STACK	81
3	PUSH 3 TO STACK	3 81
+	POP 3 AND POP 81 AND CALCULATE VALUE WITH OPERATOR (81+3) AND PUSH THE RESULT TO STACK	84
3	PUSH 3 TO STACK	3 84
/	POP 3 AND POP 84 AND CALCULATE VALUE WITH OPERATOR (84/3) AND PUSH THE RESULT TO STACK	28

Evaluate the following postfix notation of expression

a) 20 10 + 5 2 * - 10 /
b) 10 3 * 7 1 - * 23 +
c) 25 8 3 - / 6 * 10 +
d) 5 20 15 - * 25 2 * +
e) 5 3 2 4 + 5 * + 6 + -
f) TRUE OR FALSE AND NOT FALSE OR FALSE
g) TRUE FALSE NOT FALSE TRUE OR AND
h) NOT A OR B NOT B AND NOT C

Convert following Infix to Postfix Notation showing status of stack

a) X-Y/(Z+U)*V
b) A*(B+(C+D)*(E+F)/G)*H
c) (A+B*C)/D+E/(F*G+H/I)
d) A+B*C+(D*E+F)
e) (a + b - c) * d – (e + f)
f) A - B - C * (D + E / F - G) – H
g) A + ((B - C * D) / E) + F - G / H
h) (A * B - (C - D)) / (E + F)
i) A + ((B - C * D) / E) + F - G / H

Write a function in C++ to perform Push operation on dynamically allocated stack containing real number

. SOLUTION:

```
Stuct NODE
        {       Float data;
         NODE *link; }
class stack
 {
        Node *top;
        public: stack() ;
        void Push();
        void Pop();
        void display();
         ~stack(); };
```

```
void stack::Push()
 { NODE *temp;
temp=new Node;
cin>>temp->data;
 temp->link=top;
top=temp; }
```